(2)

TECHNICAL REPORT RD-BA-91-1

SOFTWARE METRICS FOR TOTAL DEVELOPMENT CYCLE EVALUATION

Ross Grable
Software Engineering Directorate
Research, Development, and Engineering Center

NOVEMBER 1991

U.S. ARMY MISSILE COMMAND
Redstone Arsenal, Alabama 35898-5000

Approved for public release; distribution unlimited.

DESTRUCTION NOTICE

FOR CLASSIFIED DOCUMENTS, FOLLOW THE PROCEDURES IN DoD 5200.22-M, INDUSTRIAL SECURITY MANUAL, SECTION II-19 OR DoD 5200.1-R, INFORMATION SECURITY PROGRAM REGULATION, CHAPTER IX. FOR UNCLASSIFIED, LIMITED DOCUMENTS, DESTROY BY ANY METHOD THAT WILL PREVENT DISCLOSURE OF CONTENTS OR RECONSTRUCTION OF THE DOCUMENT.

DISCLAIMER

THE FINDINGS IN THIS REPORT ARE NOT TO BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION UNLESS SO DESIGNATED BY OTHER AUTHORIZED DOCUMENTS.

TRADE NAMES

USE OF TRADE NAMES OR MANUFACTURERS IN THIS REPORT DOES NOT CONSTITUTE AN OFFICIAL ENDORSEMENT OR APPROVAL OF THE USE OF SUCH COMMERCIAL HARDWARE OR SOFTWARE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188
Exp. Date: Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release: distribution is unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR-RD-BA-91-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Directorate RD&E Center 6b. OFFICE SYMBOL (If applicable) AMSMI-RD-BA	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) U.S. ARMY MISSILE COMMAND Attn: AMSMI-RD-BA-AD Redstone Arsenal, AL 35898-5253	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Software Metrics for Total Development Cycle Evaluation

12. PERSONAL AUTHOR(S)
Dr. Ross Grable

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM Nov 89 TO Apr 91	November 1991	37

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Software Metrics, Software Engineering, Software Management Software Indicators

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

There are many metrics for gauging software during the several phases of development, but each leaves questions about the quality and maintainability of the software. A new metric is proposed which may overcome some of these problems while giving a strong intuitive model for gauging development progress through design, coding, and testing. A proposal is made for the collection of data for testing and evaluation of this model.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Ross Grable	22b. TELEPHONE (Include Area Code) (205) 876-3056	22c. OFFICE SYMBOL AMSMI-RD-BA-AD

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

TABLE OF CONTENTS

I. INTRODUCTION

There are many metrics for gauging software during the various activities of development, but each leaves questions about the quality and maintainability of the software. A new metric is proposed which may overcome some of these problems while giving a strong intuitive model for gauging progress through design, coding, and testing phases of development. A proposal is made for the collection of data for testing and evaluation of this model.

A. Purpose

This document serves as an updated revue of software metric literature and a proposal for developing software metric tools. It can be used as a source book for entry into current publications regarding metrics and a reference for the general content of well known metrics.

B. Application

The U.S. Army Missile Command, Software Engineering Directorate is responsible, as a Life Cycle Software Support Center, for maintenance of missile system embedded software and for a technology assessment and consultation concerning the acquisition of such software. A spectrum of metrics is required for maintenance and prediction of schedules and counseling regarding software system architectures and methodologies. This report provides insight into characteristic factors which can be and have been measured. The references give access to more detailed information.

In addition, a proposal is made to collect data from the various projects maintained by the directorate to more effectively evaluate proposals and software development efforts by contractors developing missile system software. The metric model presented is a framework for the collection and evaluation of the required information.

C. Organization

The paper first surveys management indicators, software quality data collection and measures, and software structural metrics. Details are provided concerning the factors which contribute to each metric. Calculation equations are presented where applicable. Comments are made concerning the comparison of various methods and models.

An evaluation is made of the various metrics for use in the software acquisition process. Measurable factors are identified in relation to the phase in the development cycle during which they are available and beneficial.

A unified model gives a framework for the analysis of the effects of the various factors on subsequent maintainability of the software. The model is extended to enhance the intuitive insight gained from use of the unified model.

A mapping is made between the metric factors and the structural features of the Ada programming language.

II. SURVEY OF SOFTWARE METRICS

A sizable library of software metrics is available. This brief survey will put most of the commonly used ones into perspective by describing their content and elucidating their methodologies. They are classified here as management indicators, software quality measures, and structural complexity measures.

A. Management Indicators

Software acquisition demands the use of management indicators as mandated by AR 70–13 and the standard review processes [13]. Government and contractor staff are familiar with these measures, but they are included here for completeness.

There are some good and fairly widely used methods of predicting project size, schedule, and cost. These methods are empirical, statistically based, and oriented, due to their databases, toward specific applications. For example, the Constructive Cost Model (COCOMO) [5] for computing development time says

$$MM = a(KDSI)^b m(x),$$

where MM is the number of man–months required to produce the software product, a and b are empirically derived constants obtained from production mode and level definitions, Thousands of lines of Delivered Source Instructions (KDSI) is the code, and m(x) is a factor computed from cost–driving attributes. The level and production mode parameters reflect the size and constraints required by the specific project. The cost driving attributes used to compute m(x) are:

Product attributes
- required software reliability
- database size
- product complexity

Computer attributes
- execution time constraint
- main storage constraint
- virtual machine volatility
- computer turnaround time

Personnel attributes
- analyst capability
- application experience
- programmer capability
- virtual machine experience
- programming language experience

Project attributes
- modern programming practice
- use of software tools
- required development schedule.

A derivative of COCOMO called SECOMO was developed by the Army. These measures are highly dependent on KDSI which may not be easy to estimate early in a project. None of these

factors are computed from the design structure of a project's computer programs. If there were a way to calculate KDSI from the architectural and communication structure of the early top level design, more confidence could be placed in the validity of progress reports based on KDSI dependent measures. For example, just because 90 percent of the program code is complete it may not be the difficult 90 percent.

An approach to measuring predicted size comes from function point analysis [30], a metric based on both software characteristics and environment [4]. The size of a program module in lines of source code is computed as

$$SIZE(SLOC) = (ARCH)(EXPF)((LANG*FP_A)+OOC_N)^a,$$

where the factors are defined as:

ARCH = architectural factor
EXPF = expansion factor
LANG = language expansion factor
FP_A = adjusted function point count
OOC_N = normalized operand/operator count.
a = reuse factor.

Each of the factors has a defined range and is adjusted in magnitude for the specific application being sized. Typically, the architectural factors would be defined as:

centralized	1.0
tightly coupled multiprocessor	1.3
loosely coupled multiprocessor	1.5
federated	1.6
distributed with central database	1.8
fully distributed	2.1
array processor	0.9

The expansion factor EXPF is a product,

$$EXPF = c_k \prod_{i=Q}^{n} SM_i,$$

where c_k is a calibration factor and SM_i is a size modifier which can be defined typically as:

requirements volatility	.95 to 1.18
database size	.94 to 1.11
degree of real time	.90 to 1.16
use of modern programming techniques	.93 to 1.11
use of software tools	.89 to 1.10
analyst capability	.89 to 1.19
application experience	.91 to 1.15
environment experience	.95 to 1.10
language experience	.91 to 1.13.

LANG for the language Ada would be 72 lines of code per function point with a correlation of about .887 for Reifer's data set. The function point count FP_A is the sum of inputs, outputs, master files, modes, and interfaces. For real time systems, the sum would include stimulus–response pairs and rendezvous.

Function point analysis depends indirectly on the internal structure of the source code. It has reportedly [30] been successful in measuring 28 projects within 20 percent and has been made accessible through desk top computers. But, the internal structure of modules is not visible enough to determine whether the most complex or difficult work on a project has been completed or even well defined.

There are other methods in addition to COCOMO for measuring effort as surveyed by [9]. These include:

1. criteria based on validity, objectivity, ease of use, sensitivity, transportability, and other subjective top level qualities,
2. methods based on least squares curve fit to parameterized linear and non–linear functions of time which represent effort level,
3. level of difficulty models which are calculations based on subjective estimation of level of difficulty of various phases of development,
4. statistical models using regression analysis to fit polynomials or other curves, such as the Rayleigh distribution, to data on software development time,

All of these management indicators are intended to provide top–level and not a detailed view of the software and [1] its development. A survey of management indicators shows measurements for computer resource utilization, software development effort, requirements and definition stability, software progress, development and test, cost/schedule deviations, and the use of software development tools.

B. Software Quality Measures

As can be seen from a general survey of the literature on software metrics [10], there are:

1. classic metrics based on software science, cyclomatic complexity, and function points,
2. life cycle metrics based on analysis, software design, code structure, quality assurance, and method,
3. code metrics designed for specific languages,
4. new metrics based on the development process and effort, graph structure of software, and information content,
5. metrics based software process models.

From the abundance of software measuring methods there should come better metrics. The ideal metric should be an automatable one based on the architectural and communication structure of the abstract software system such that progress in the development process can be measured and an intuitive judgement can be made as to the complexity and quality of the resulting software [24,28].

Many metrics of software quality and style indicators use internal properties of code, some being very subjective and others being automated and analytical. Most of these require source code for analysis, and thus cannot be initiated in the design phases of a project. But, these metrics give valuable insight into what should be measured in order to determine the internal quality, complexity, and developmental progress of software. In contrast to management indicators, quality indicators have a higher level of resolution with respect to the internal characteristics of the software.

Generally, the highest level attributes have been related to low level characteristics of software, but not in a quantitative way. For example, [31] shows the relationships demonstrated in Figure 1. As Rossan points out, design indicators are measured, using some common scale, from programs and documentation citing attributes present in the software. Management indicators are the results of reviews, inspections and tests, and software behavior using behavioral and acquisitional metrics. What is needed is a set of design indicators which can be used as the basis for management indicators of a more meaningful nature.

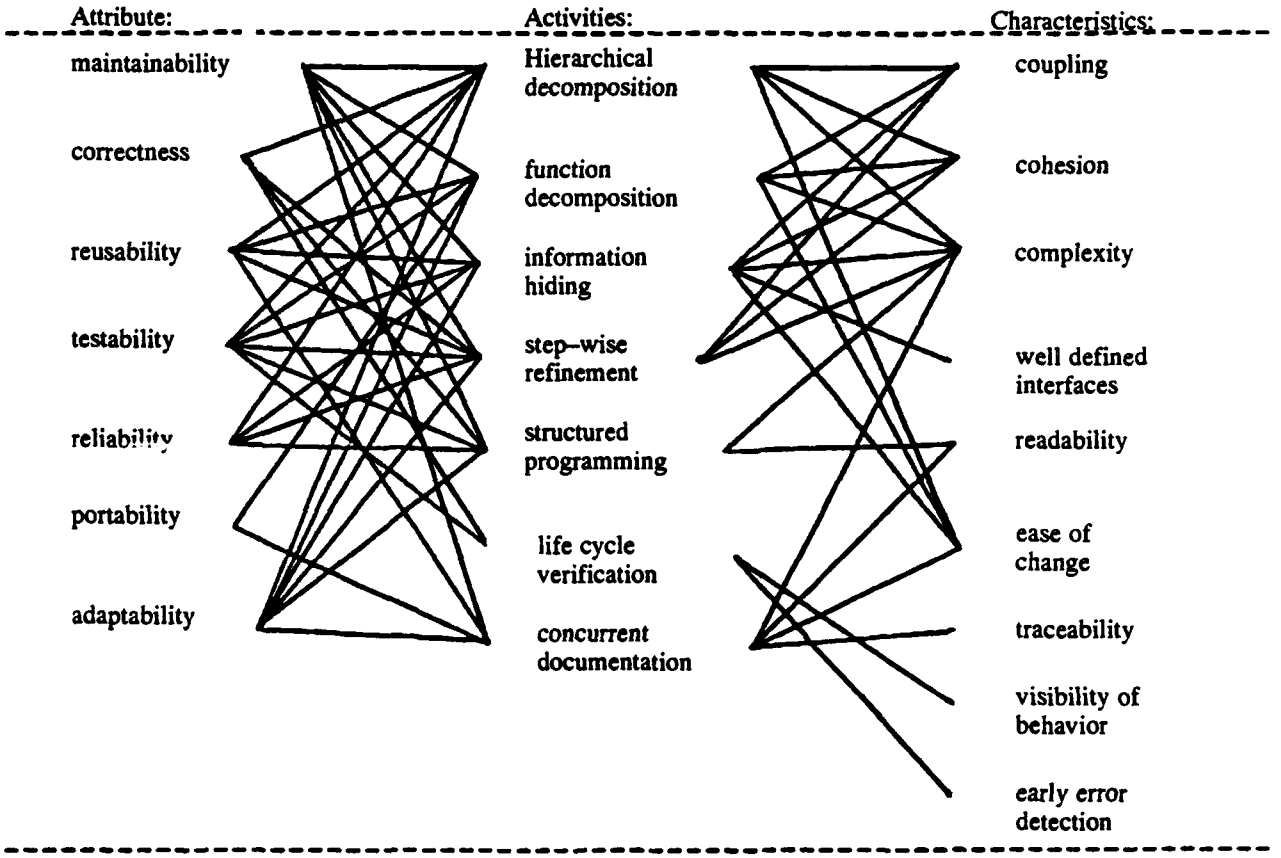


Figure 1. Attribute to Characteristic Mapping.

Another approach to the organization of metrics for software quality is given by [3]. A quality metric tree, shown here in indentured form, shows the relationships of higher level properties to lower level software characteristics:

Quality

correctness

completeness, consistency, traceability

efficiency
concision, execution efficiency, operability
flexibility and maintainability
complexity, concision, consistency, expandability, generality, modularity, self documentation, simplicity
integrity
auditability, instrumentation, security
interoperability
communication, commonality, generality, data commonality modularity
portability and reusability
generality, hardware independence, modularity, self documentation, software system independence
reliability
accuracy, error tolerance, simplicity, consistency, modularity
testability
auditability, instrumentation, self documentation, simplicity, complexity, modularity
usability
operability, training

The lower level characteristics specified in this model can not be measured economically with today's technology. Nevertheless, all of these factors should contribute in some way to the evaluation of a software product. Even though, as pointed out by [11], it is difficult to have a metric which can measure both process and product, there should be a way for these factors to provide feedback to developers and programmers as the project unfolds.

One way to provide such feedback is the work sheets which result from reviews. In fact, work sheets [34] and check lists [29] often provide a principle medium for measuring software quality. Manuals have been written [26] which present trade–offs generated by software standards and delineate quality factor rating guidelines based on subjective work sheets. Integrity, maintainability, portability, reusability, usability, testability, flexibility, and interoperability are all at odds with program code efficiency in the traditional sense. Flexibility, interoperability, and reusability must be balanced with the code integrity. Reusability must be measured with respect to reliability. Factors such as these can be measured subjectively using worksheets and quality factor rating guidelines. Factors and metrics used in such evaluations are typically [26] quality of comments, complexity, completeness, operability, user interface, data commonality, effectiveness of comments, traceability, consistency, training, and communication commonality. Check sheets generate values which form a matrix for k modules and n module level measurements as

$$M_d{}^m = \begin{matrix} m_{1\,1} & m_{1\,2} & \cdots & m_{1\,k} \\ \vdots & & & \vdots \\ m_{n\,1} & & \cdots & m_{n\,k} \end{matrix}$$

From this matrix, indicators can be calculated such as, for metric i, the average and distance from mean would be

$$A_i = \Sigma_{j=1}^{k} (M_{ij}/k)$$

$$\sigma_i = \Sigma_{j=1}^{k} (M_{ij} - A_i)^2/k \ .$$

Based on the result, a module j would be reported for examination if

$$M_{ij} < A_i - \sigma_i \ .$$

Factors in this model can be normalized by data from previous projects. Thus, resolution and flexibility for application are available using these methods, but, a more objectively based metric would be preferable.

Another aspect of software quality, which must be measured for the purpose of software acquisition, is supportability. A supportability metric has been developed by Frank Blackwell of the Army Missile Command's Software Engineering Directorate. This model is based on the COCOMO model [5]. The Subsystem supportability factor is calculated by summing the values for each factor selected in a supportability matrix and then subtracting the total from 100. A subsystem scoring the highest supportability in each factor will receive a rating of 100 (excellent). A subsystem scoring nominal in each factor will receive a rating of 75 (fair). Any subsystem scoring below 60 is considered to have unacceptable supportability. Also, a subsystem can be rated as having unacceptable supportability if the Memory or Throughput Utilization exceeds a specific value. The overall subsystem suppportability average is calculated by multiplying each of the subsystem supportability ratings by the subssystem source lines of code, totaling the scores, and then dividing by the total source lines of code. The four system supportability factors are then totaled and added to the subsystem average for the subsystem supportability rating. If a subsystem is determined unsupportable, the system supportability should be calculated without the subsystem and the unsupportable subsystem should be identified. The model can be used to determine deficient areas and areas where improvement will result in a higher supportable rating. The supportability factors and their definitions are shown in Figure 2.

Indirect approaches to quality measurement have been used such as examination of the design documentation as the primary data source. Taxonomies have been developed to aid in such documentation analysis. The following documentation tree [33] shows many factors common to the quality of the program code itself.

adequacy

- **accuracy**
  - requirement/design traceability
    - (top down, bottom up equivalence)
- **consistency**
  - conceptual
    - (invariance of concept)
  - factual
    - interface, database security, error recovery, I/O, performance, timing

completeness

domain coverage

document relationships

decomposition (refinement enunciation)

referential

TBD/TBS, % missing, % appropriate

modification tracking

code

documents

usability

logical traceability

references (TBD/TBS, missing, appropriate)

term consistency

sufficiency of index and table of contents

intra–document completeness

readability (consistency, standards)

physical (print, format, modularity)

accessibility/availability

expandability

In this model, the recurring themes of measurable upper level supported by subjectively measurable lower level parameters appear. The dependence on secondary characteristics introduces another level of abstraction away from measuring the actual software architecture.

To some extent the quality of software can be determined by the nature of the fault structure which emerges during testing and integration. Fault analysis is an important part of measuring software quality and reliability. Standard metrics for fault content [20] are:

1. fault density per Thousand Lines of Source Code (KLOC),
2. defect density per KLOC based on defect found in reviews,
3. cumulative failure profile
4. fault–days and various combined defect indices.

Fault analysis is a two edged sword in that the more faults collected for the analysis the more valid the model is; yet, more faults (disregarding seed faults) undermine confidence in the system. One type of fault analysis which can be easily automated is the determination of variables defined but not used [36]. This method is based on predicate calculus models of requirements and leads to specification–dependent testing of software. The system generates test data and programs. In this case, an evaluation metric is not the end product. Another example of a fault analysis system produces random inputs independently from the input domain according to a typical operation distribution. Errors produced are counted and analyzed using deterministic Baysian and Markov error counting models. Although automated tools can be used to collect fault data, the major problem remains in not having the metrics early enough in the project's development.

Intuitively, design quality can contribute to fewer faults for correction in the end product. In view of this, measurement of specific attributes of the software resulting from a design would be helpful. An example of such an approach is [2]. This particular method

collects data on the properties of developed program code to determine whether a given development methodology can generate high quality code. The factors used in this analysis are described by the following tree:

Reusability

- Hierarchical decomposition
- Information hiding
  - coupling
  - cohesion
  - well defined interface
    - globals, passed parameters, execution coupling
    - data structure coupling, parameterless calls
  - ease of change
  - complexity
- Functional decomposition
- Concurrent documentation

The goal was to design an automated system for computing the quality. There are automated tools commercially available which claim to calculate numbers for source code quality. One such tool is ADAMAT. Unfortunately, the metric is proprietary making analytic evaluation difficult. Another commercial tool is called Logiscope from the French company Verlog [35]. It collects and analyzes statistics from source code of programs. This package uses the metrics of Halstead, McCabe, and Mohanty discussed later in this paper.

In order to summarize standard metrics being used, the IEEE has published a standard dictionary of software measures [20] which presents a collected reference list of descriptions of useful metrics. The list of Table 1 categorizes the measures into seven groups. The number in parentheses is the number of the measure as it appears in the dictionary. Measures preceded by I are intensive and those preceded by E are extensive in nature.

TABLE 1. Software Measures.

Based on faults

I	(1)	Fault Density
I	(2)	Defect Density
E	(3)	Cumulative Failure Profile
E	(4)	Fault–Days Number
I	(8)	Defect Indices
E	(9)	Error Distribution
I	(11)	Manhours Per Defect
I	(20)	Mean Time To Discover the Next K Faults
I	(21)	Purity Level
I	(22)	Estimated Number of Faults Remaining (by seeding)
I	(27)	Residual Fault Count
I	(28)	Failure Analysis by Elapsed Time
I	(29)	Testing Sufficiency
I	(30)	Mean Time to Failure
I	(31)	Failure Rate
I	(36)	Test Accuracy
E	(38)	Independent Process Reliability

TABLE 1. Software Measures (continued).

Based on Requirements

I	(5)	Functional or Modular Test Coverage
I	(6)	Cause and Effect Graphing
I	(7)	Requirements Traceability
I	(10)	Software Maturity Index
E	(12)	Number of Conflicting Requirements
E	(17)	Minimal Unit Test Case Determination
I	(23)	Requirements Compliance
I	(24)	Test Coverage
I	(35)	Completeness

Related to Test Design

I	(5)	Functional or Modular Testing Coverage
E	(17)	Minimal Unit Test Case Determination
I	(18)	Run Reliability
I	(24)	Test Coverage
I	(26)	Reliability Growth Function

Related to Variable Counts

E	(14)	Software Science Measures
E	(25)	Data or Information Flow Complexity

Related to Software Structure

E	(13)	Number of Entries and Exits per Module
E	(15)	Graph–theoretic Complexity for Architecture
E	(16)	Cyclomatic Complexity
E	(32)	Software Documentation and Source Listings

Based on Performance

E	(37)	System Performance Reliability
I	(39)	Combined Hardware and Software Operational Availability

Based on Management Parameters

I	(33)	RELY (Required Software Reliability)
I	(34)	Software Release Readiness

In terms of internal software attributes, there are several important factors to determine in measuring software quality. Table 2 shows some factors and how they might be measured during the various phases of software development. Experimentation must determine which factors correlate with errors and maintenance and to what extent. There is large potential for significant work in this area. As pointed out in the previous sections of this paper, several factors and metrics have already been identified as significant.

TABLE 2. Sources for Metric Data.

factor \phase	HMCD	CMCD	DD	PDL	CODE	TEST
number of modules accessing						
data variable	C	C	C	C	C	AEC
control variable	C	C	C	C	C	AEC
data structure	C	C	C	C	C	AEC
subprogram	–	C	–	C	C	AEC
exception	–	C	–	C	C	AEC
compilation unit	–	C	–	C	C	AEC

factor \phase	HMCD	CMCD	DD	PDL	CODE	TEST
items accessed by a module						
data variables	EST	C	C	C	C	AEC
control variables	EST	C	C	C	C	AEC
internal variables	–	–	C	C	C	AEC
internal data struct	–	C	C	C	C	AEC
data structures	C	C	C	C	C	AEC
files	C	C	C	C	C	AEC
subprograms	EST	C	–	C	C	AEC
rendezvous	C	C	–	C	C	AEC
exceptions	–	C	C	C	C	AEC
compilation units	–	C	–	C	C	AEC
states	–	–	–	C	C	AEC
operation modes	EST	C	–	C	C	AEC
execution paths	–	–	–	C	C	AEC
requirements met	C	C	C	C	C	AEC
algorithm determinacy	EST	EST	–	EGT	EGT	ICB
commentary description	ICB	ICB	–	EGT	EGT	–
Valid loop termination	–	–	–	–	ICB	ICB
lines of code	EST	EST	–	EST	C	–

factor \phase	HMCD	CMCD	DD	PDL	CODE	TEST
variables' parameters						
scope	–	TSH	CPA	TSH	TSH	AEC
value range	–	–	CPA	CPA	CPA	AEC
type	–	–	EGT	EGT	EGT	EGT
type variegation	–	–	C	C	C	C
access mechanism	–	–	EGT	EGT	EGT	AEC
containing structure	–	TCS	TCS	TCS	TCS	AEC
effect (data, control)	–	EGT	EGT	EGT	EGT	AEC
exceptions	–	C	C	C	C	AEC
machine dependencies	–	–	ICB	ICB	ICB	AEC
initial value	–	–	ICB	ICB	ICB	ICB

TABLE 2. Sources for Metric Data (continued).

units/scale	–	–	EGT	ECI	ECI	AEC
error containment	–	EGT	EGT	ECI	ECI	AEC
validity proveability	–	–	–	ECI	ECI	AEC
loop termination	–	–	–	ICB	ICB	ICB
name length	–	–	ICB	ICB	ICB	–
requirement item	C	C	C	C	C	ICB
format	–	–	EGT	EGT	EGT	ICB
internal cohesion	–	–	EGT	EGT	EGT	–
volatility	–	–	EGT	EGT	EGT	AEC
commentary description	–	–	ICB	EGT	EGT	–

Abbreviations:

column headings:

HMCD	high level module communication diagram
CMCD	complete module communication diagram
DD	data dictionary
PDL	program development language document
CODE	source code for the computer program
TEST	test results statistics report

methods of measurement

AEC	count in actual execution
C	count all potential occurrences
CPA	count possible occurrences in range
ECI	evaluate computational impact
EGT	evaluate graded types
EST	use pre–defined estimation factor
ICB	increment a count of boolean values
TCS	trace and count containing structures
TSH	count total through sub–hierarchy
–	measurement not appropriate

In the design phase, most of the evaluation must be subjective. Initial software module communication diagrams can be used to establish and evaluate encapsulation and abstraction of variables, data structures, external devices and subprogram units. The Hrair limit limiting the number of structures at any one level can be enforced by counting modules. The initial graphs can be used to aid construction of a requirements traceability matrix and a data dictionary which contain much more precise data about the system communication architecture. As the system architecture is recursively refined, more precision is available for measurable quantities. The use of development tools will greatly enhance not only the collection of data for metrics but also the enforcement of development standards which can improve system reliability.

When designing and writing programs, the programmer/analyst is aware of specific attributes which add or reduce the chaos or improve the probability of success of the software as an entity. Low scores are desirable for the m(x) factor in the COCOMO model, low scores are desirable for function point parameters, high ratings are desirable for the "–ilities" of sub-

jective software quality, low probability of faults is needed, high scores on commercial metrics are desired, and low complexity metric values are sought. In order to measure the quality of software, key elements from the usable indicators need to be extracted and distilled into a well formulated, consistent, and validatable model. To achieve coveted values for metrics, the code must adhere to certain standards which are measurable from the design and code of the software. For some examples, it must

1. exhibit a low degree of coupling between modules
2. have a highly coherent algorithmic structure
3. demonstrate good encapsulation of data structures, procedures, and functions, with proper exception handling,
4. use well defined typing of variables including value limits, and initialization,
5. have a standard well formed architecture,
6. demonstrate adequate commentary,
7. use only portable features of the implementation language.

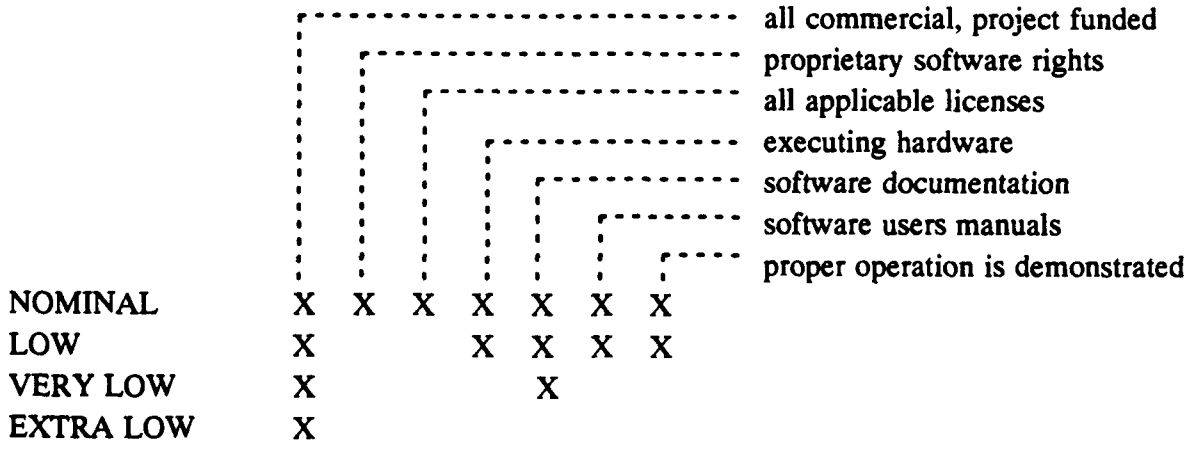
None of the methods seen thus far can measure, from design through test, the internal quality, complexity, and progress of computer program development. In other words, the standard measures for software fall short of being able to measure effectively the factors which contribute to difficulty of creation and maintenance. There should be an automatable software metric designed which can integrate quality and architectural factors into an intuitively interpretable dynamic gauge of the product.

C. Supportability Factors

1. Distinct programming languages – The number of distinct programming languages utilized in the system

2. Distinct Architectures – The number of distinct hardware architectures utilized in the system. Families of processors, such as the 680X0, are considered a single architecture.

3. Delivery – The level of Life Cycle Software Support Environment hardware, support software, and documentation delivered.



4. Software Management – the level of software configuration management, software quality management, and management insight into the software development process to ascertain satisfactory development progress and status accounting.

	software quality assurance independent of the software development project management, formal procedures to assure periodic management review of the status of the: software development process, mechanisms in place for assuring that software subcontractors follow a disciplined software development process, formal configuration management of the tactical and support software,	coding standards, internal independent verification; and validation, and software development indicators.	Informal software configuration management, limited software quality assurance, and minimal design reviews
HIGH	X	X	
NOMINAL	X		
LOW			X

5. Subsystem Size – The size of the software subsystem. The subsystem size factor takes into account the size, in lines of code, of the subsystem software and the implementation language. ADA HOL – Software programmed in Ada and compiled on a validated Ada compiler. NON–ADA STANDARD HOL – Software programmed in a standard high order language other than Ada.SPECIAL APPLICATION ASSEMBLY LANGUAGE – Software programmed in assembly language for a special purpose processor.

6. Design Complexity – The complexity of the software subsystem.

7. Memory Utilization – The program instruction and data storage memory utilization of the target processor(s).

8. Throughput Utilization – The throughput utilization of the target processor(s).

9. Program Design Language (PDL) Implementation – The PDL used in developing the subsystem software.

HIGH – An Ada format PDL which can be successfully compiled by a validated Ada compiler is used during the design of the software.
NOMINAL – A non–Ada PDL is used in designing the software.
LOW – No PDL is used in designing the software.

10. Processor Type – The type of processing element which executes the subsystem software.

HIGH – The software is executed on a commercial computer.
NOMINAL – The software is executed on a standard general purpose processor.
LOW – The software is executed on a special purpose processor.
VERY LOW – The software is executed on a processor developed specifically for the application.

11. Computer Turnaround Time – The computer response time of the software support environment (compile time, etc.). TURN: Computer Turnaround Time.

12. Modern Programming Practices – The degree to which modern software

engineering and programming practices are used in developing the software. MODP: Use of Modern Programming Practices.

13. Tools – The degree to which automated software tools are used in developing the software subsystem.

TOOL: Tool support.

	¦	¦	¦	¦	development tools commercially available
		¦	¦	¦	on call maintenance is available
			¦	¦	maintenance contract is available
				¦	support is available
					custom tools
VERY HIGH	X	X	X	X	
HIGH	X		X	X	
NOMINAL	X			X	
LOW				X	X development tools obsolete. No support.

14. Software Documentation – The level of software documentation developed for the software subsystem.

	¦	¦	¦	¦	Approved Standard Software Documentation
		¦	¦	¦	Independent Verification and Validation
			¦	¦	documentation includes software software design documents, software test plans, procedures, and reports,
			¦		requirements specifications, product specification.
				¦	contractor format.
HIGH	X	X	X	X	
NOMINAL	X		X	X	
LOW				X	
VERY LOW					X

15. Software Testing – The level of software testing performed during software development.

	¦	¦	¦	¦	Approved Standard Software Testing
		¦	¦	¦	Independent Verification and Validation
			¦	¦	testing includes unit, major components, system integration testing
				¦	customer witnessing of the Formal tests
				¦	Internal contractor procedures.
HIGH	X	X	X	X	
NOMINAL	X		X	X	
LOW				X	
VERY LOW					X

D. Structural Complexity Measures

There are factors and characteristics of the software which are known very early in the design process and which can be measured throughout the development process. The focus of effort should be on identifying those factors and integrating them into a validatable metric based on software internal properties yet reflecting quality and development progress at the management level. There is some research leading in this direction.

There are two classical metrics of code structure. Software science [17] uses the number of distinct operators n1, the number of distinct operands n2, the total number of operators N1, and the total number of operands N2 to calculate several characteristics such as

program vocabulary	=	l	=	$n1 + n2$
observed length	=	L	=	$N1 + N2$
estimated length	=	L'	=	$n1(\log_2 n1) + n2(\log_2 n2)$
volume	=	V	=	$L (\log_2 n)$
difficulty	=	D	=	$(n1/2)(N2/n2)$
program level	=	L1	=	$1/D$
effort	=	E	=	$V / L1$
number of errors	=	B	=	$V / 3000 = E^{2/3} / 3000$
alternate length	=	L''	=	$\log_2 ((n1)\,!\,) + \log_2 ((n2)\,!\,)$

of which volume is most often cited. The other classical metric is cyclomatic complexity [25] in which a directed linear graph of the software execution path structure is analyzed. The graph is made strongly connected by joining end to beginning. Then, N is the number of nodes, E is the number of edges, SN is the number of splitting nodes, RG is the number of regions of the resulting graph. The complexity C is calculated as

$$C = E - N + 1 = RG = SN + 1.$$

Both of these metrics are often used as a baseline for evaluating other metrics because data are relatively easy to obtain.

A most notable effort in the right direction is the work on software metrics based on information flow [18, 19]. This method uses a lexical approach to measuring system connectivity. Passed parameters and accesses to global data stores are tallied to give the number of inputs and outputs for each module. Lines of source code are also tallied. The complexity of a module is calculated as

$$\text{complexity} = (\text{lines of code}) * (\text{inputs} * \text{outputs})^2.$$

The lines of code estimate turns out to be a non–critical factor and does not need to be precise. In fact, it was noted that the length factor may detract from the accuracy of the metric.

Thus, the complexity, calculated, reveals several things about the structure of the code. For example, if the number of inputs and outputs is high, the module may be implementing more than one elementary function. Large input*output can indicate stress points in the system because more effect on the system is indicated. Inadequate refinement of modules also leads to large I/O product. Thus, high complexity for a module may not indicate a specific problem but can show that there is a problem. The complexities of individual modules are linearly summed to give the complexity of subsystems and systems.

Information flow metrics correlate well with change records of UNIX operating system maintenance [22]. In comparing with other standard complexity metrics, this method does quite well. Correlation factor with system errors was .95 for information flow, .96 for Halstead, and .89 for McCabe. Halstead and McCabe correlated at .84 but information flow correlated .38 with Halstead and .35 with McCabe. This indicates a fair degree of orthogonality between information flow and the classical metrics of Halstead and McCabe.

Another metric [14] uses an abstract state machine description of the software. A functional basis is used for state definitions. Links are then established in semantic nets which reflect the requirements. The net links objects, sets and actions embodied in the system. The state machines are described in three ways; by enumeration listing the states, by axiomatics listing logical conditions characterizing the system, and by algorithmic analysis defining range and domain. In this case, system analysis is made manageable by a hierarchical decomposition of the system. The algorithmic paradigm provides a way to extend the state description to mathematical methods. The most difficult part of the analysis is the mapping of requirements to specifications. However, the method does reduce ambiguity by delineating explicit relations in a defined context.

Structure and style contribute to software quality and should be factored into metrics. For example, encapsulation of data, procedures, functions, and data structures can play an important role in the successful development of a software project particularly in avoiding lurking side effects. A lurking side effect is one of those little illicit data item changes that jump out to byte the unsuspecting software maintainer/user in the most embarrassing moment. The varying degrees of coupling between software modules can have varying degrees of impact on the architectural structure. The quality of cohesion within a module effects the conceptual complexity of the module and should in some way be reflected in a quality metric. Yet, all criteria for good software cannot be measured. Indeed, there are some software standards which should be accepted as minimum attributes requiring no measurement.

There are issues critical to software development and quality which are not addressed by the available metrics. Computability and formal verification form entire disciplines malignant with active research. A great amount of maturity will be required in these areas before relevant metrics can be integrated into system evaluation.

There are then many holes in the software metric picture. Obviously, that is why much research is being published. Perhaps there is a way that many diverse properties and concerns related to success of software systems can be brought together under a single simple but powerful intuitive model which generates some useful metrics. A concept which did a similar service in thermodynamics and information theory is the concept of entropy.

III. A UNIFIED PARADIGM FOR SOFTWARE METRICS

Rather than relying on several metrics at various respective stages of software development, this section describes a model by which some of the measures can be integrated into a single one based on a physical analogy.

The concept of entropy has been suggested to interpret the development of software as a process of reducing entropy of the system design [21]. Evaluation of hardware complexity has also been done [23]. Notable use of an entropy measure has been done in evaluating software design [27]. Entropy has been used as a metric for software complexity relative to cellular array machines [15]. These uses of the concept of entropy are obviously different and not to be confused, but the analogies are very useful intuitively when used with care.

In physics, change in entropy S can be defined as an integral of the reciprocal of temperature T with respect to the differential of heat dQ. For a reversible process with volume changing from V_1 to V_2, the change in entropy S_1 to S_2 is

$$S_2 - S_1 = \int dQ/T = k(\ln V_2 - \ln V_1),$$

giving entropy a basically logarithmic form [16]. The base of the logarithm and the constant k determine the units of measure. In communication theory, the logarithm base is 2 so that the unit of measure is the bit. From communication theory [32, 7], entropy is expressed as

$$S = -\Sigma_i \, p_i \, \log_2 p_i,$$

where p_i is the probability of message i occurring and

$$\Sigma_i \, p_i = 1.$$

In software design evaluation, entropy is expressed as

$$H(P_1, \ldots, Pn) = \Sigma_i^n = 1 \frac{|x_i|}{|x|} \log_2 \frac{|x|}{|x_i|},$$

where x_1 , ... , x_n are distinct classes of subsystems called and

$$P_i = | x_i | \; / \; | x |$$

is the probability of subsystem x_i being activated. For software complexity on cellular array machines, entropy is defined as the maximum performance factor over the array $\max_a PF_c$ times the number c of cells in the array or

$$S = c^* \max_a PF_c.$$

The performance factor PF_c is defined in terms of SPF_{ci}, the product of the number of states used times Hamming distance between binary state use vectors, and LPF_{ci}, the product of the number of communication links used times the Hamming distance between binary link use vectors, as

$$PF_c = \Sigma_i \log_2 (1 + SPF_{c\,i} + LPF_{ci}).$$

None of these metrics provide what is needed for software tracking. However, by nature, entropy should be linearly summable; yet the combinatoric nature of error probabilities suggest that factors should be combined multiplicatively. Therefore, the traditional logarithmic nature seems correct. What, then, will be the precise form and what are the contributing factors for software entropy?

For software, entropy should indicate in some sense the possibility of an error occurring. Many factors contribute to errors such as typographical errors, variables exceeding boundaries, logic errors, shear size of the project, the nature of the language used, syntax errors, etc. Reducing the possibility of error, reducing entropy, is done in different ways at different times in the development process. For example, during requirements specification, limits and performance parameters must be clearly spelled out keeping in mind the development methodology to be used. In the initial design phase, entities, operations, and communication must be formulated to minimize entropy by using sound architecture and methodology. During testing, each test should lower system entropy by proving doubtful constructs in the code and verifying that requirements are met. When errors are found in testing, entropy increases. New tests must be designed to again lower entropy to pre–error–detection levels. Thus, to measure entropy, different data must be used in different phases of a project yet scale factors must be included for compatibility of the metric throughout the project. The process would be structured as shown in Figure 2.

The basic form for software entropy is

$$SWENT = \Sigma_j\, c_j\, [\, \Sigma_i\, (\, (1/T) \log (sf^p)\,)_i\,]_j,$$

where c_j is an empirical constant based on the software development phase j under analysis. The external sum is taken over the various phases. The internal sum is taken over the individual modules i of the software. The factor T reflects the quality of design usage of entities such as variables, subprograms, and data structures. The factors reflects the number of operational states or modes of the module i. The factor f measures the interface size for the module. The cyclomatic complexity for the state structure is p. Each of these factors will be described in detail.

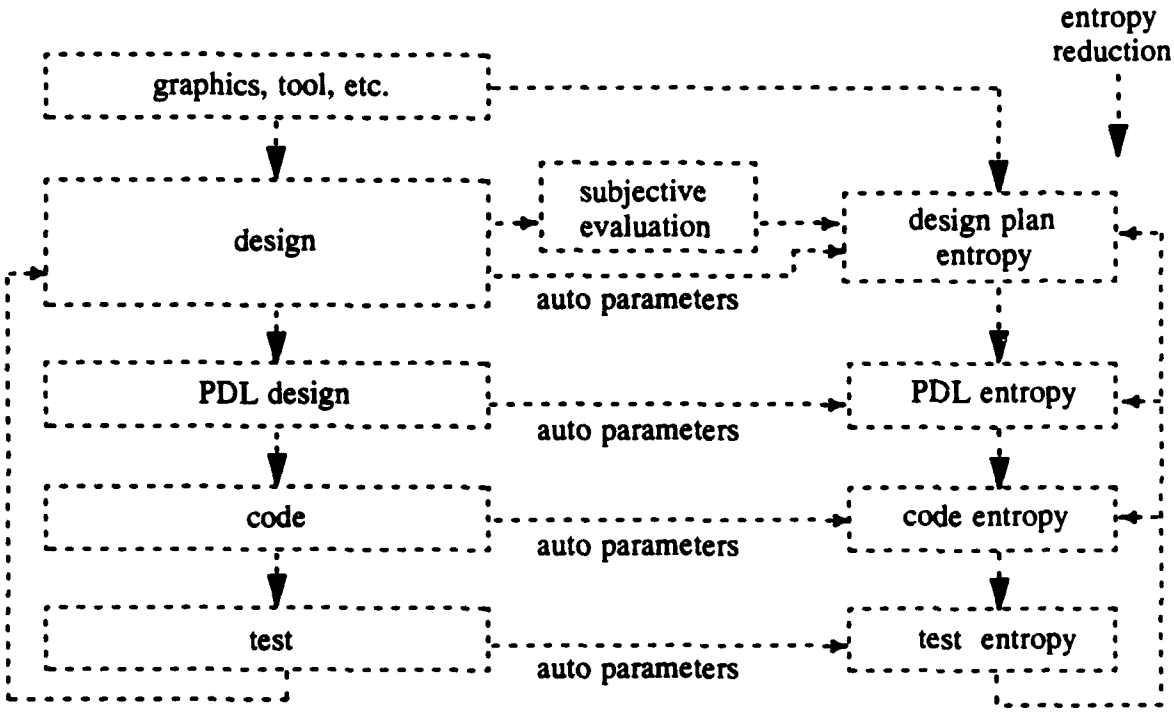


Figure 2. Entropy Metric Cycle.

For the initial highest level design phase, the constant c would be relatively large due to the uncertainties of measurements and estimates at that stage. It must also be large since the number of measurable factors is rather small yet the factors which can be measured have potential for high impact on system entropy. In the later stages where source code data are available, c would be relatively small. This is the case because more factors will be summed and because each factor has potentially less relative impact on system entropy. The constant c is negative for the test phase because testing will decrease the system entropy indicating an increase in confidence in the software. The entropy of a system will never be zero or negative so the choices of the constants c_j should be chosen appropriately. Each time new design or revisions are made, entropy will increase and will have to be brought back down by more testing.

The factor T is called "influence" and is computed as

$$T = \Pi_j \ \Pi_i \ \ (w_i \ m_{ij})^{x},$$

where the product is made over all attributes i for all of the specific entities j being evaluated. Influence, T, is intended to be a gauge of the effects of encapsulation and abstraction of entities within the sphere of influence of the module being considered. The entities for which T is calculated for each module includes all variables, data structures, subprograms, and external utilities referenced by the given module. The factor w_i for each attribute i is a weighting factor which is empirically determined. w is higher for attributes which have higher correlation with module success. Success in this sense is determined by faults detected in systems for which data has been collected. The exponent x is –1 or +1 depending on whether the attribute is advantageous or detrimental to software quality. The factor m_i for each attribute i is the actual

measure of the attribute. The attributes and their measurement methods are shown in Table 3. If the factor i does not apply to a given entity, the value of w_i m_i is unity so that T is not effected.

TABLE 3. Code Characteristics Factors.

Attribute	i	x	Measurement
Scope/ the scope of accessed entitlies	1	– 1	count modules accessing
Value Range/	2	+ 1	boolean existence of specified range
is the range of value	3	– 1	numberic range relative to median
restricted and how	4	– 1	enumeration cardinality
Access/ difficulty extracting an item from a structure	5	– 1	number of nodes traversed to value
Coupling Effect/ depth of coupling effect of the item	6	– 1	value = 1, control = 2, rendezvous = 3
Exception Scope/ depth of resolution	7	– 1	count subprogram levels to resolve
Hardware Dependency/	8	– 1	boolean (numeric range, type, format)
Initialization/ initial value defined	9	+ 1	boolean
Scaling Required/ use of scale factors	10	– 1	float = 1, integer = 5, interface = 20
Error Estimate/ impact of calculation errors	11	– 1	% relative numeric error introduced
Loop Termination/ loop control effect	12	– 1	boolean yes for loop terminators
Name Length/ encourage meaning in naming entities	13	+ 1	count characters in name
Requirement Item/ defined in system specification	14	+ 1	boolean existence of traceability
Internal Cohesion/ classes of data controlled by entity	15	– 1	count classes of data items

TABLE 3. Code Characteristics Factors (continued).

Type/	16	+ 1	existence and variegation booleans
unique type declared			
Validity/	17	+ 1	boolean: provably valid
provably valid calculation for value	18	– 1	probability of invalid result
Volatility/	19	– 1	value changes per operation cycle
changability of the entity	20	– 1	changes in entity definition
Commentary/	21	+ 1	characters in descriptive comments

The factor f, called "interface", measures the impact of the module through inputs, outputs and, possibly superfluously, the size of the module. This factor has been studied and validated to a certain extent [18]. It is calculated as

$$f = loc*(inputs*outputs)^2 = 1\,(io)^2\,,$$

where "loc" is the number of lines of source, "inputs" is the number of inputs to the module and "outputs" is as expected. It has been shown that loc may not be significant [ibid].

The factor s, called "states", is a count of the states or modes resident in respective modules of software. That is

$$s = \text{number of states.}$$

This measure may be rather subjective, but it can be calculated from an enumeration typed variable which can define the state of the software module. This factor harks back to the usage of machine state definitions of software complexity [14, 8, 15,].

The factor p, called "paths", is the log2 of count of the execution paths in the algorithms of a module. For example, paths may be counted in certain types of algorithms as 2^b where b is the number of binary branch decision statements or as the number of linear independent execution paths in the Cyclomatic Complexity sense.

During the testing phase of software development, the entropy should decrease. This happens through the combination of p, the number of paths tested, and the factor c_j being negative. As errors are discovered in testing, entropy terms are added back in for that module until testing can bring the figure back down to the normal decreasing trend for testing phase.

The calculation of entropy includes the factors most important for gauging software quality and progress. One remaining problem with the use of this entropy as a metric for software quality and progress is that it is not an absolute quantity and a relativity base must be established for given types and sizes of applications. A graph of the progress of entropy trends during the development of a project has more meaning than a single abstract value at a single point in time. The main advantage is the ability to track software as it becomes more refined. Comparisons can be made between software projects which use the metrics consistently. Entropy variability throughout the software project gives a basis for intuitive understanding of progress. Also, reliability and fault tolerance issues can be addressed using this concept of entropy.

IV. APPLICATION OF METRICS TO THE ADA ENVIRONMENT

The structure of the language Ada contributes particularly well to the use of software engine metrics. The elements required for calculating T, s, f, and p are explicit in Ada program code. Not only that, software engine metrics, through the mechanism of decreasing entropy in the system, can encourage the use of constructs provided in Ada explicitly for increasing the reliability and maintainability of software [6].

A. Mapping to the Structural Features of Ada

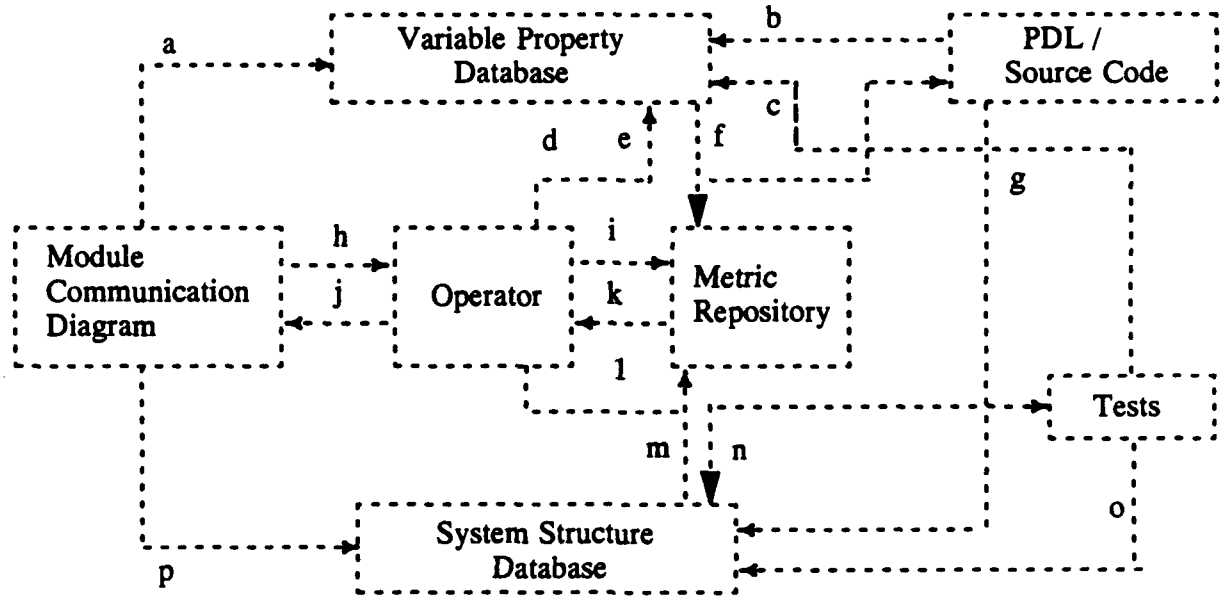
Encapsulation of variables, subprograms, data structures, data types, etc. are explicit. Data items are strongly typed, specific in range or by enumeration, initializable, restricted in scope, and over–loadable with explicit exeception handling. Machine dependencies are traceable through types, number sizes, and pragmas. Compileable modules are explicit in packages with the ability to abstract structures and subprograms through the use of private types. Reuse is encouraged through the explicit use of generic subprograms. Communication of parallel procedures is explicit through task rendezvous. Applications of the software engine metrics are more difficult with other languages because of differences in available compilers. However, given that the measure collection tool is compatible with the language compiler, the metrics can be collected and used for tracking the progress and quality of the software product as shown in Figure 3.

The Module Communication Diagram is a module which encapsulates and abstracts the graphics used to specify the highest level design of the software system. It operates in the earliest phases of software development, requests acceptance of data and sends, to the System Structure Database and the Variable Property Database, information which it has extracted from the design graphics. The Variable Property Database is a repository for information about the properties of variables and data structures in the system. The System Structure Database is a repository for information about the structure of the software. The Operator is an external interface which allows control of metric gathering system. The PDL/Source Code module encapsulates and abstracts the collection of data from the program design language and program code. Operating in the software design and coding phases, it requests access and sends data to the Variable Property Database and the System Structure Database. The Tests module performs the analogous task through the software testing phase of the development. The Metric Repository collects data from the Variable Property Database and the System Structure Database, calculates statistics for metric validation and computes the metrics. The results are output at the request of the operator.

Several commercial and public domain data collection and metric calculation systems are available such as LOGISCOPE [35], ADAMAT [12], and tools available from the National Ada Repository.

B. Development of a Validation Database

The Software Engineering Directorate (SED) of the U.S. Army Missile Command's Research, Development and Engineering Center is responsible for developing guidelines to insure that software acquired for missile systems is maintainable. The loop closes when SED becomes responsible for the maintenance of the software. In this configuration, SED has access to actively maintained systems and can collect metrics data it needs for acquisition. Some research has been done by SED staff in this field.



a. accept message data type
b. accept variable data
c. accept value data
d. collect data
e. send variable data
f. variable data
g. accept structure data
h. level available
i. set operation phase
j. collect data
k. metric results
l. collect data
m. structure data
n. send structure data
o. accept path data
p. accept module relation data

Figure 3. Ada Data Extraction System.

The scope of effort for a useful study would follow a standard structure for such projects. The outline of tasks appears as follows:

1. Design systems for data collection regarding
   a. design attributes
   b. code structure
   c. maintenance hours per module.
2. Select systems for analysis.
3. Collect data on relevant system components such as
   a. design attributes
   b. structure attributes of code
   c. personnel hours of maintenance per module.
4. Correlate design and structure data with maintenance data.
5. Analyze and summarize experimental results.
6. Write and publish results

7. Design application systems for use of metrics in acquisition of missile system software.

The data collection and analysis effort would concentrate on the parameters required for validation of the Software Engine Metrics model as well as standard metrics which can be further validated. Raw data should be formatted in a manner which gives high resolution access to the raw measurement process.

REFERENCES

1. AMC–P–70–13, 1987, "Army Materiel Command Software Management Indicators", U.S. Army Materiel Command, Alexandria, VA.

2. Arthur, James D., and Nance, Richard E., 1987, "Developing an Automated Procedure for Evaluating Software Development Methodologies and Associated Products", Technical Report SRC–87–007, Systems Research Center and Department of Computer Science, Virginia Tech, Blacksburg, VA, U.S. Navy Systems Research Center BOA N60921–83–G–A165–B023.

3. Arthur, Jay, 1984, "Software Quality Measurement", Datamation, Vol.30, (Dec 15).

4. Behrens, Charles A., 1983, "Measuring the Productivity of Computer Software Development Activities with Function Points", IEEE Transactions on Software Engineering, Vol. SE–9, no.3 (Nov), p 648–651.

5. Boehm, Barry W., 1981, Software Engineering Economics, Prentice–Hall, Englewood Cliffs, NJ.

6. Booch, G.,1983, Software Engineering with Ada, Benjamin/Cummings, Menlo Park, CA.

7. Brillouin, L., 1962, Science and Information Theory, Academic Press, New York , NY.

8. Britcher, Robert N., and Gaffney, John E., 1982, "Estimates of Software Size from State Machine Design", Seventh Annual Software Engineering Workshop Report SEL–82–007, December.

9. Conte, S.D., Dunsmore, H.E., and Shen,V.Y., 1985, "Software Effort Estimation and Productivity", Advances in Computers, Marshall C. Evicts bed., Vol.24, BB., Academic Press, Orlando, FL.

10. Cote,V., Burgle, P., Ogling, S., and Revert, N., 1988, Software Metrics: An Overview of Recent Results, J. Systems and Software, Vol. 8, No.2 (Mar), pp. 121–131.

11. Curtis, B., 1981b, "The Measurement of Software Quality and Complexity", Software Metrics: An Analysis and Evaluation, A. Perlis et.al. eds, MIT Press, Cambridge, MA.

12. DRC , 1987, ADAMAT Users Manual, Dynamics Research Corp., Andover, MA.

13. Fenick, S., 1990, "Implementing Management Metrics: An Army Program", IEEE Software, Vol. 7, No.2, (MAR), pp. 65–72.

14. Ferrantino, A.B. and Mills, H.D., 1977, "State Machines and Their Semantics in Software Engineering", IEEE COMSAC, Chicago, Fall.

15. Grable, D.R., 1989, Cellular Array Machines and Their Behavior Under Micro–Dataflow Regimes, Ph.D. dissertation, Department of Computer Science, University of Alabama in Huntsville, Huntsville Alabama.

16. Halliday, David, and Resnik, Robert, 1978, Physics, John Wiley, NY.

17. Halstead, M.H. 1977, Elements of Software Science, Northholland, New York.

18. Henry, S. and Kafura, D.H., 1981, "Software Structure Metrics Based on Information Flow", IEEE Trans. on Software Engineering, Vol SE–7, No.5 (Sept), p. 510.

19. Henry, S. and Selig, C., 1990, "Predicting Source Code Complexity in the Design Stage", IEEE Software, Vol.7, No.2, (Mar), pp. 36–44.

20. IEEE, 1988, IEEE Standard Dictionary of Measures to Produce Reliable Software, Std 982. 1–1988, IEEE, New York.

21. Jensen, R.W. and Tonies, C.C., 1979, Software Engineering, Prentice Hall, Englewood Cliffs, NJ.

22. Kafura, D. and Henry, S.,1981, "Software Quality Based on Interconnectivity", J. for Systems and Software Vol.2, pp. 121–131.

23. Kulch, W., 1972, "Entropy of Transformed Finite State Automata and Associated Languages", in Graph Theory and Computation, Read, R.C. ed. Academic Press, New York, NY.

24. Li, H.F., and Cheung, W.K., 1987, "An Empirical Study of Software Metrics Compared", IEEE Trans. on Software Eng., Vol. SE–13, No.6 (June).

25. McCable, T.J., 1976, A Complexity Measure, IEEE trans. Software Engineering, Vol. SE–2 (Dec), pp. 308–320.

26. McCall, James A., and Matsumoto, Mike T., 1980, "Software Quality Measurement Manual", RADC–TR–80–109, Vol 2, Rome Air Development Center, Griffin AFB, NY.

27. Mohanty, S.N., "Entropy Metrics for Software Design Evaluation", J. Systems and Software, Vol. 2, pp. 39–46.

28. Perlis, A., Sayward, F., and Shaw, M. eds, 1981, Software Metrics: An Analysis and Evaluation, MIT Press, Cambridge, MA.

29. Poore, J.H., 1988. "Derivation of Local Software Quality Metrics", Software Practical Experience, Vol. 11 (Nov) pp. 1017–1027.

30. Reifer, Don J., Function Point Analysis, Reifer Consultants, Inc., Torrance CA.

31. Rosson, C.V., 1988, "Management Indicators: Assessing Product Reliability and Maintainability", Technical Report SRC–88–011, Systems Research Center, Virginia Tech, Blacksburg, VA, Naval Sea Command contract N60921–83–G–A165–B031.

32. Shannon, and Weaver, A Mathematical Theory of Communication, U. of Illinios Press.

33. Stevens, K. Todd; Arthur, James D. and Nance, Richard E., 1988, "A Taxonomy for the Evaluation of Computer Documentation", Technical Report SRC–88–008, Virginia Polytechnic Institute and State University, Blacksburg, VA, Naval Surface Warfare Center Contract N60921–83–G–A165–B038.

34. System Architects, Inc., 1982, "Introduction and General Instructions for Computer System Acquisition Metrics Handbook Volume I", ESD–tr–82–143 (I), Electronics System Division, Air Force Systems Command, Hanscom Air Force Base, MA.

35. Verlog, 1988, LOGISCOPE User's Guide, Verlog, Alexandria, VA.

36. White, Lee J., 2987, "Software Testing and Verification", in Advances in Computers, Marshall C. Yovites ed, Vol. 26, p. 335, Academic Press, Orlanda, FL.

DISTRIBUTION LIST

	Copies
AMSMI–RD,	1
AMSMI–RD–CS–R	15
AMSMI–RD–CS–T	1
AMSMI–RD–BA, Bill Craig	1
AMSMI–RD–BA–AD	1
AMSMI–RD–BA–AD, Dr. Ross Grable	10
AMSMI–RD–BA–PD	1
AMSMI–RD–BA–SE	1
AMSMI–QA–QT–SP	1
AMCPM–AM–H, Tony Pollard	1
AMCPM–AM–H, Lex Scott	1
AMSMI–GC–IP, Mr. Fred H. Bush	1
U.S. Army Materiel System Analysis Activity ATTN: AMXSY–MP (Herbert Cohen) Aberdeen Proving Ground, MD 21005	1
IIT Research Institute ATTN: GACIAC 10 W. 35th Street Chicago, IL 60616	1 1
Software Engineering Institute Carnegie–Mellon University Pittsburg, PA 15213	1
Director, Technical Information Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209	1
Defense Technical Information Center Cameron Station Alexandria, VA 22314	1
Commander, U.S. Army Material Command Attn: AMCQA 5001 Eisenhower Ave. Alexandria, VA 22333–0001	1

	Copies
Commander, U.S. Army Natick Rd&E Center Attn: STRNC–EP Natrick, MA 01760–5014	1
Commander, U.S. Army LABCOM Attn: AMSLC–PA Adelphi, MD 20783	
Commander, U.S. Army Communication Electronics Command Attn: AMSEL–PA Ft. Monmouth, NJ 07703–5000	1
Commander, U.S. Army Aviation Systems Command Attn: AMSAV–Q 4300 Goodfellow Blvd. St. Louis, MO 63120–1798	1
Commander, U.S. Army Test and Evaluation Command Attn: AMSTE–AD Aberdeen Proving Ground, MD 21005	1
Director, U.S. Army Management Engineering College Attn: AMXOM–DO Rock Island, IL 61299–7040	1
Commander, U.S. Army Operational Test & Evaluation Command Attn: CSTE–ESE–S Park Center, 4501 Ford Ave. Alexandria, VA 22302–1458	1
Commander, U.S. Army Health Care System Support Activity Attn: Mr. Jack Huffman Bldg. 2000 Ft. Sam Houston, TX 78234–6050	1
Commander, U.S. Army Material Command Attn: AMCDE–CS, Mr. Howard Kea 5001 Eisenhower Ave. Alexandria, VA 22333	1
U.S. Army Aberdeen Proving Ground Attn: STEAP–IM–AL Aberdeen Proving Ground, MD 21005–5001	1
Director, U.S. Army Ballistic Research Lab Attn: AMXBR–OD–ST Aberdeen Proving Ground, MD 21005–5066	1
HQ, U.S. Army TECOM Attn: AMSTE–TO–F Aberdeen Proving Ground, MD 21005–5055	1

	Copies
Commander, U.S.A. Communications & Electronics Command Attn: AMSEL–ME–PSL Ft. Monmouth, NJ 07703–5007	1
HQ, U.S. Army CECOM Attn: AMSELL–LG–JA Ft. Monmouth, NJ 07703–5010	1
Commander, U.S. Army Aviation Systems Command Attn: AMSAV–DIL 4300 Goodfellow Blvd, East 2 St. Louis, MO 63120–1798	1
Commander, U.S. Army Missile Command Attn: AMSMI–RD–CS–R Redstone Arsenal, AL 35898	1
Commandant, U.S. Army School of Engineering & Logistics Attn: AMXMC–SEL–L Red River Army Depot Texarkana, TX 75507–5000	1
DIRECTOR U.S. Army TRADOC Systems Analysis Acty Attn: ATAA–SL (Tech Library) White Sands Missile Range, NM 88002	1
U.S. Army Aviation School Library P.O. Drawer 0 Ft. Rucker, AL 36360	1
USMA Attn: Mr. Egon Weiss, Librarian West Point, NY 10996	1
Commandant U.S. Army Engineering School Attn: Library Ft. Belvior, VA 22060	1
U.S. Army Humphey's Engr. Spt. Activity Attn: Library Branch Ft. Belvior, VA 22060	1
Engineer Topographic Lab Attn: STINFO Ft. Belvior, VA 22060	1

	Copies
Pentagon Library Attn: Chief, Reader's Service Branch The Pentagon, Room 1A518 Washington, DC 20310	1
U.S. Army Corps of Engineers Attn: DAEN–ASI–Tech Library 20 Massachusetts Ave. NW Room 3119 Washington, DC 20314	1
U.S. Army Operational Test & Evaluation Agency Attn: Tech Library 5600 Columbia Pike, Room 503 Falls Church, VA 22401	1
Naval Mine Warfare Engineering Activity Code 322 Yorktown, VA 23691	1
Commander Naval Facilities Engineering Command Attn: Library 200 Stovall St. Alexandria, VA 22332	1
David W. Taylor Naval Ship RD&E Center Library Division, Code 5220 Bethesda, MD 20084	1
Naval Air Systems Command Attn: Tech Library Air 00D4 Washington, DC 20361	1
Naval Surface Weapons Center Attn: Tech Library Dahlgren, VA 22448	1
Naval Research Lab Attn: Tech Library Washington, DC 20375	1

	Copies
Naval Surface Weapons Center Attn: Tech Library Silver Springs, MD 20910	1
Naval Sea Systems Command Library Documentation Branch Sea 9661 Washington, DC 20362	1
Naval Ship System Engineering Station Technical Library Code 011F Bldg 619 Philadelphia, PA 19112	1
Naval Training Equipment Center Attn: Technical Library Orlando, FL 32813	1
HQ, USMC Marine Corps Technical Library Code LMA–1 Washington, DC 20314	1
Air Force Systems Command Technical Information Center HQ AFSC/MPSLT Andrews AFB, DC 20334	1
Defense System Management College Attn: Library Bldg 205 Ft. Belvior, VA 22060	1
Director, Defense Nuclear Agency Attn: TITL Washington, DC 20305	1